CORE JAVA MODULE 1 NOTES :

24 June 2025 12:28

➢ Module 1: Java Basics & Environment Setup

No.	Topic
1.1	What is Java? History, Features, Real-World Use & Honest Comparison with Python, MERN, C++
1.2	Platform Independent, JDK vs JRE vs JVM
1.3	JVM (Java Virtual Machine) full Architecture
1.4	Installing JDK & Setting up IDE (VS Code / IntelliJ / Eclipse)
1.5	Your First Java Program – Step-by-step Hello World
1.6	Understanding Compilation and Execution (javac + java)
1.7	Interview Questions and Assignment

➢ Module 1.1 – What is Java?

1. Real-Life Example:

✧ **"Imagine This…"**
→ You're a student going to **hostel**.
→ You pack a **charger**, but guess what?
→ Your friend has an iPhone, someone else has a Samsung, another one has an old Nokia phone.
→ Now everyone is like:
→ **"Bhai charger de naaa!"**
→ And you're like: **"Yeh toh sab pe nahi chalega!"** 😵

Wouldn't it be cool if you had **one universal charger** that worked for **ALL phones**?

✧ **That universal charger = Java**

2. Programming World Analogy:

→ Java is like that **universal charger** in the tech world.
→ You write your Java code **once** and it runs on **any computer** (Windows, Mac, Linux) without rewriting or recharging!
→ This is what made Java so powerful:
✓ **"Write Once, Run Anywhere"** – also called **WORA**.

3. Concept Notes :

✧ **What is Java?**
- Java is a **high-level, object-oriented programming language**.
- It is **platform-independent**, meaning:
  Code written in Java can run on any device having a **Java Virtual Machine (JVM)**.
- Java is not just a language — it's also a **platform** and **a whole ecosystem**.

✧

Java Platform	JVM + JRE + JDK = Code likho aur chalao
Java Ecosystem	Frameworks, Libraries, Tools, Community

4. History of Java (With Fun Flavour):

Year	What Happened
1991	A group at Sun Microsystems (called the **Green Team**) wanted to make software for TVs, remotes, etc.
1995	They made a language called **Oak** 🌳. But the name was taken, so they renamed it **Java** ☕ (after Java coffee!)
2009	**Oracle** bought Sun Microsystems and became the new daddy of Java

Now Java powers **Android, Banking, E-commerce, Gaming, NASA, etc.**

➢ Features of Java (With Real-Life Examples)

Feature	What It Means	Real-Life Example
Platform Independent	Code runs anywhere	Like Maggi — banalo stove pe, microwave pe, induction pe
Object-Oriented	Everything is an object	Like PUBG — Players, Guns, Bullets = Objects
Simple	Easy syntax, no confusion like pointers	Like using Google Maps vs old paper maps
Secure	No virus can misuse your memory	Like OTP before UPI payment
Robust	Crash-resistant, handles errors	Like a cricket helmet — protects you even if hit
Multithreaded	Runs multiple tasks together	Like eating chips while watching Netflix
High Performance	Fast due to JIT Compiler	Like a scooty with turbo booster
Distributed	Supports networking apps	Like Zomato app – location, food tracking
Dynamic	Loads classes during runtime	Like your teacher entering the class unexpectedly

4. Real-World Usage of Java

Java is everywhere! I will realize how powerful it is:

Sector	Java Usage
Banking (SBI, HDFC)	Backend for transactions, ATMs
Android	Java was the official language for Android development
E-commerce	Amazon, Flipkart backend systems
Cab Services	Ola/Uber ride-tracking, billing logic
Gaming	Games like Minecraft are built in Java
Space Research	NASA uses Java for simulation software
Education	University management systems, portals

➢ Java vs Python vs MERN vs C++ (Honest Comparison Table)

Real-Life Example :

🎬 **"College Admission Analogy"**

Imagine you're choosing a college...

- ✓ One has **big campus, strong placements, but strict rules** (Java)
- ✓ One has **cool crowd, easygoing professors, but less industry exposure** (Python)
- ✓ One has **startup vibe, fast growth, but jugaadu management** (MERN Stack)

You think: **"Kaunsa choose karu?"**

Same with programming languages!

Each one has strengths. But today, you'll understand **why Java might be YOUR best choice.**

Language	Strengths	Weaknesses	Best Use
Java	Platform-independent, Secure, Fast, OOP, Robust, Used in real-world systems	Verbose syntax, learning curve(It takes **more time to understand** than Python)	Android, Banking, Large Systems
Python	Easy syntax, Fast prototyping, AI/ML popularity	Slower execution, weak mobile apps, less strict typing	AI/ML, Scripting, Quick tools
MERN (JS)	Full web stack, Fast dev, Startup trend	Less secure, messy for large projects	Web apps, Startups
C++	Fast, close to hardware, good for performance-heavy systems	complex, not cross-platform friendly	Gaming, Embedded Systems

→ Java may not be the “coolest” or “shortest” language —
→ But it's like **Virat Kohli** — consistent, strong, industry-tested.
→ Where Python is “easy to start”, Java is “built to last.”

➢ Where Java Is MORE Powerful (Real World)

1. **Enterprise-Grade Projects**

– Banks, insurance, government portals run on Java (e.g., SBI core systems)

2. **Android App Development**

– Native Android used to be purely Java (now mostly Kotlin, but still Java-based)

3. **Backend for Large Systems**

– Flipkart, Amazon, Netflix backend = Java + Spring Boot

4. **Security + Performance**

– JVM and strong typing make it secure and robust for critical apps

5. **Job Market & Longevity**

– Java has been in **top 5 languages** for over 20 years in **TIOBE & StackOverflow** indexes

6. Where Java Falls Short (Be Honest)

Point	Weakness
◆ Verbose Code	◆ Even a simple "Hello World" needs a class, main method, etc.
◆ Not Best for AI/ML	◆ Python wins in data science libraries like TensorFlow, Pandas
◆ Slower Prototyping	◆ Python or JS faster for quick ideas
◆ Android Shift	◆ Android now encourages Kotlin (though it's Java-based)

➢ 7. Should YOU Learn Java?

✓ YES — if you want:

- A strong career in **backend development**, **enterprise software**, **Android**, or **full-stack with Spring Boot**
- To **understand OOP, memory, architecture** deeply
- **Job security + long-term growth**
- **Clear core logic** to transition into any language later

NOT IDEAL — if you want:

- To just build quick tools, ML scripts, or experiments
- Fancy frontend design/web games
- Extremely short learning curve

➢ 8. Summary:

- Java is a **strong foundational language** — like learning driving on a manual car. Once you master it, auto is a piece of cake.
- You can **shift to Python, MERN, etc. later easily** — they’ll feel easier after Java.
- Most **real-world job systems are still built in Java**.
- If you want to become a **serious developer** (backend, Android, full-stack), **Java is a great starting point**.

➢ Motivation Line

“Languages may come and go... but Java? Java is like your reliable best friend — not flashy, but always there when things get serious.”

⇒ Module 1.2 – JDK vs JRE vs JVM (With Real-Life + Code Analogy)

1. Real-Life Analogy: Movie Theater Experience

Imagine this:
You go to watch a movie

- 🎬 The movie = Java **Program (Code)**
- The actor = **Compiler** (converts movie script to performance)
- Projector screen = **JVM** (executes the movie)
- Ticket + Snacks = **JRE**
- Behind-the-scenes tools like camera, mic, editing software = **JDK**

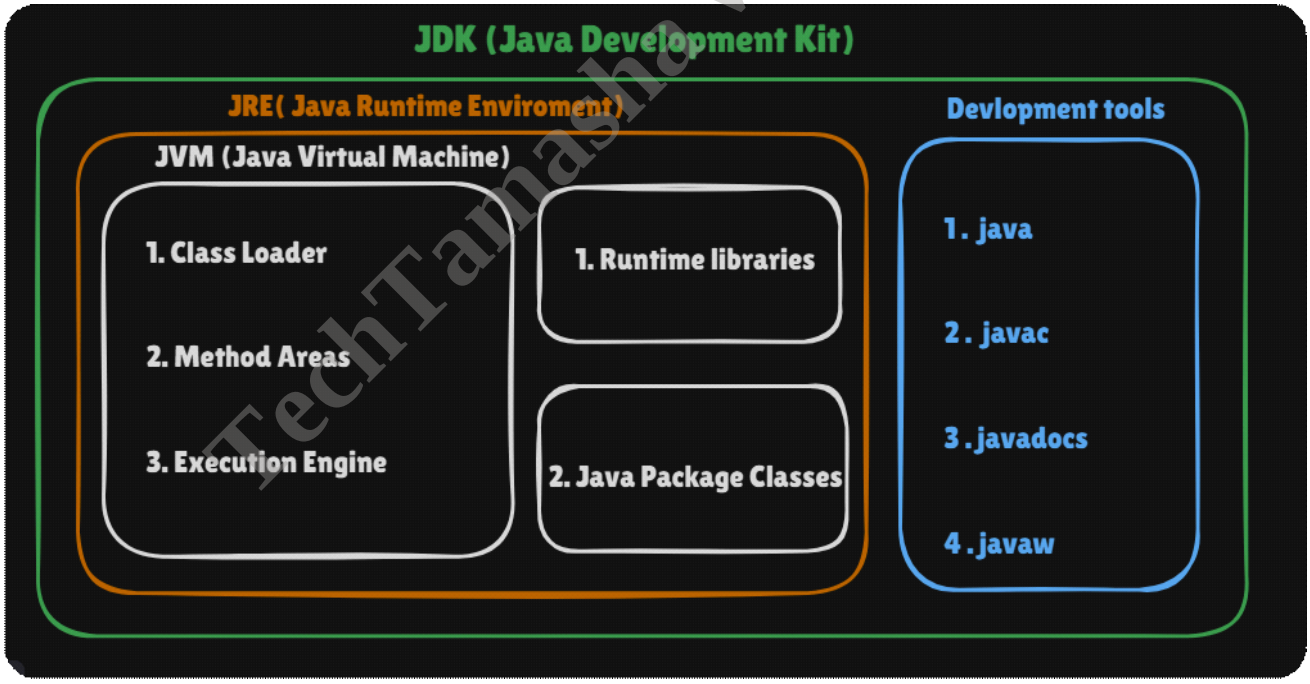
So...

- You **make** a movie using the **JDK**
- You **watch** a movie using the **JRE**
- And the **JVM** is the magic screen where it actually runs!

2. Programming Analogy: The Java Delivery Chain

Component	Meaning	Analogy
◆ **JDK**	◆ Java Development Kit	◆ Kitchen where the chef (dev) prepares food
◆ **JRE**	◆ Java Runtime Environment	◆ Dining room where food is served
◆ **JVM**	◆ Java Virtual Machine	◆ Waiter who brings the food to table

- You can't **serve food** (run Java program) without JRE
- You can't **cook food** (write/compile code) without JDK
- JVM makes sure everyone gets the same taste — Windows, Mac, or Linux!



➢ JDK (Java Development Kit)

- ◈ It is the **full development package** to build Java applications.
- ◈ Contains everything in the JRE **+ compiler (javac), debugger, tools, etc.**
- ◈ Needed for **writing, compiling, and running** Java code.

➢ JVM (Java Virtual Machine)

- ◈ It is the **engine** that actually runs the Java program.
- ◈ **JVM is responsible for converting bytecode (.class file) into machine code**, line by line.
- ◈ It provides important features like **garbage collection, memory management, and security**.

➢ **JRE (Java Runtime Environment)**

- ◈ It is a **package that contains JVM + libraries + other runtime files** required to run Java applications.
- ◈ **It does NOT contain compiler or development tools** – it is only for **running** Java programs.

⇒ Module 1.3 : JVM Architecture – Simple Explanation with Fun Example: "College Exam Center" :

🎓 *"Imagine JVM is like a whole College Exam Center system, where students (your Java code) come to give an exam, get verified, allocated seats, write the exam, papers are checked, and finally results are declared!"*

◈ 1. Class Loader Subsystem = College Admin Desk:

📋 *"Every student entering the exam center must first go through registration!"*

What it does:
Loads .class files (your compiled Java programs) into JVM memory.

Real-Life Analogy:

- **Student = .class file**
- **Class Loader = Entry Gate Admin**
- They check your hall ticket (is this a valid class?), assign you to a block (memory)

➢ **Three Types of Class Loaders:**

Loader Type	Definition	Real-Life Analogy	Example
1. Bootstrap Class Loader	Loads **core Java classes** from the rt.jar (like java.lang, java.util) — the foundation of every Java app.	*Government-allotted compulsory classes (English, Math)*	Automatically loads String, System, Object, etc.
2. Extension Class Loader	Loads **optional Java extensions** from the ext folder (like javax.sound, javax.crypto).	*Extra department electives like Music or Drawing*	If your app uses audio or security extensions
3. System (Application) Class Loader	Loads **your program's classes** from the classpath (your packages, libraries).	*Subjects you selected for exam prep from school list*	Loads your custom CustomerService.java, Product.java etc.

◈ 2. Linking Phase = Checking Student Eligibility

After loading, JVM **verifies and prepares the class** before execution.

Phase	What It Does	Analogy
◆ **Verification**	◆ Ensures .class file is valid, safe	◆ Teacher checks admit card, ID
◆ **Preparation**	◆ Allocates space for static variables	◆ Assign student roll numbers
◆ **Resolution**	◆ Converts symbolic links to actual memory	◆ Map student's name to seating position
◆ **Initialization**	◆ Executes static blocks & assignments	◆ Actual data given: int x = 10

◈ 3. Runtime Data Areas = Exam Hall Memory

Different areas in JVM memory where student activity is managed during the exam

Area	Purpose	Analogy
◆ **Method Area**	◆ Class-level info (blueprints)	◆ Whiteboard showing exam rules
◆ **Heap**	◆ Stores objects, data	◆ Big storage shelf with answer sheets
◆ **Java Stack**	◆ Stores current thread's method calls	◆ Student's desk – personal
◆ **PC(Program counter) Register**	◆ Holds next instruction to execute	◆ Eye on the next question number

◆ **Native Method Stack**	◆ For native (C/C++) code	◆ Special desk for students using calculator
◆ **Constant Pool**	◆ Stores constants(String), symbols(method name , field name , class name)	◆ Question bank references

❖ Example of java stack:

➢ Scenario: Final Exam in a College Hall

- You have **3 students**: Anjali, Rahul, and Sneha
- Each one is **solving their own question paper**
- Each student has:
  - Their own **desk** (= Java Stack)
  - Their own **question flow** (= Thread)
  - Their own **rough paper stack** (= Method Calls + Local Variables)

4. Execution Engine = The Student Giving the Exam:

This is where bytecode is actually executed:

Component	Job	Analogy
◆ **Interpreter**	◆ Executes one instruction at a time (line by line)	◆ Student solving 1 question at a time
◆ **JIT(just in time) Compiler**	◆ Converts repeated instructions into fast native code	◆ Student starts using shortcut tricks
◆ **Code Cache**	◆ Stores compiled native code	◆ Student notes common answers in margin

❖ What is Code Cache in JVM?

Definition:

- **Code Cache** is a special memory area where the JVM stores **compiled native code** after it converts Java bytecode into **machine code** using the JIT compiler.

In Simple Words:

- JVM **doesn't always interpret bytecode line-by-line** — it eventually **compiles frequently-used methods** into **fast machine code**, and stores that in the **Code Cache**.

This is done by the **JIT (Just-In-Time) Compiler**.

✧ ***Program to show optimization:***

```
public class JITSimple {
   public static void main(String[] args) {

       // Warm-up
       for (int i = 0; i < 10000; i++) run();

       // Before JIT
       long t1 = System.nanoTime();
       for (int i = 0; i < 1000; i++) run();
       long t2 = System.nanoTime();
       System.out.println("Interpreter: " + (t2 - t1)/1_000_000.0 + " ms");

       // Trigger JIT more
       for (int i = 0; i < 1000000; i++) run();

       // After JIT
       long t3 = System.nanoTime();
       for (int i = 0; i < 1000; i++) run();
       long t4 = System.nanoTime();
```

```
        System.out.println("JIT:         " + (t4 - t3)/1_000_000.0 + " ms");
    }

    static int run() {
        return 10 + 20; // Simple code, still optimized
    }
}
```

🔷 5. Garbage Collector = Exam Hall Peon

Cleans up unused objects from memory (heap).

- Peon comes, picks up unused pages, erasers, etc.
- Prevents mess in the hall

🔷 6. JNI (Java Native Interface) = Student Asking for Calculator

- If a student needs to do a calculation that's **not allowed by default**, he asks invigilator.
- Similarly, JVM may call **native (C/C++) code** using JNI.

🔷 7. Security Manager & Bytecode Verifier = CCTV + Invigilator

- Ensure no cheating (malicious code)
- Prevents code from accessing files it shouldn't

⇒ Module 1.4 : Installing JDK & Setting up IDE (VS Code / IntelliJ / Eclipse)

1. Install JDK (Java Development Kit) – Common Step for All IDEs

□ Steps:

1. Go to: https://www.oracle.com/java/technologies/javase-downloads.html
2. Download the latest **JDK 17 or JDK 21** (LTS preferred).
3. Choose your OS:
   - Windows .exe
   - Mac .dmg
   - Linux .tar.gz
4. Install it like any normal software (Next → Next → Finish).
5. **Verify installation**:
   - Open terminal/command prompt:

     ```
     java -version
     javac -version
     ```

If both show a version, you're ready.

Option 1: Setting up IntelliJ IDEA

Steps:

1. Go to: https://www.jetbrains.com/idea/download
2. Download **IntelliJ IDEA Community Edition** (Free).
3. Install it (Next → Next → Finish).
4. Open IntelliJ → Select:
   - **New Project**
   - Choose **Java**
   - Make sure it shows JDK path (if not, click **Add JDK** and select it from C:\Program Files\Java\jdk-XX)
5. Project setup:
   - Name: FirstJavaProgram
   - Template: Java Hello World (optional)

➢ **Run Hello World:**

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello Java World 🌍");
```

```
  }
}
```
Click Run → Output will be shown in the bottom console.

Option 2: Setting up Eclipse IDE

Steps:

1. Go to: https://www.eclipse.org/downloads/
2. Click **Download Eclipse IDE for Java Developers**.
3. Run installer → Select Eclipse IDE for Java Developers → Install.
4. Launch Eclipse → Choose a **workspace** (folder where your projects will be saved).
5. Create new project:
   - File → New → Java Project
   - Project Name: FirstJavaProject
   - Click Finish
6. Right-click src → New → Class
   Name: HelloWorld, check public static void main.

Write Code:

```
public class HelloWorld {
   public static void main(String[] args) {
      System.out.println("Hello Java from Eclipse!");
   }
}
```

Right-click → Run As → Java Application → Output shown below.

⇒ Module : 1.5 – Your First Java Program (Step-by-step Hello World)

Step 1: Create a File

Filename: HelloWorld.java
(The file name **must match** the class name)

Step 2: Write the Code

```
public class HelloWorld {
   public static void main(String[] args) {
      System.out.println("Hello, Java World 🌍");
   }
}
```

Step-by-Step Explanation

Line	Code	Explanation
1	public class HelloWorld	Start of the program. Think of class as a **container** or **box** for your code.
2	{	Start of the class body.
3	public static void main(String[] args)	This is the **entry point** of every Java program. JVM looks for this line to start execution.
4	{	Start of main method body.
5	System.out.println("Hello, Java World 🌍");	This prints text on the screen. Like saying something aloud.

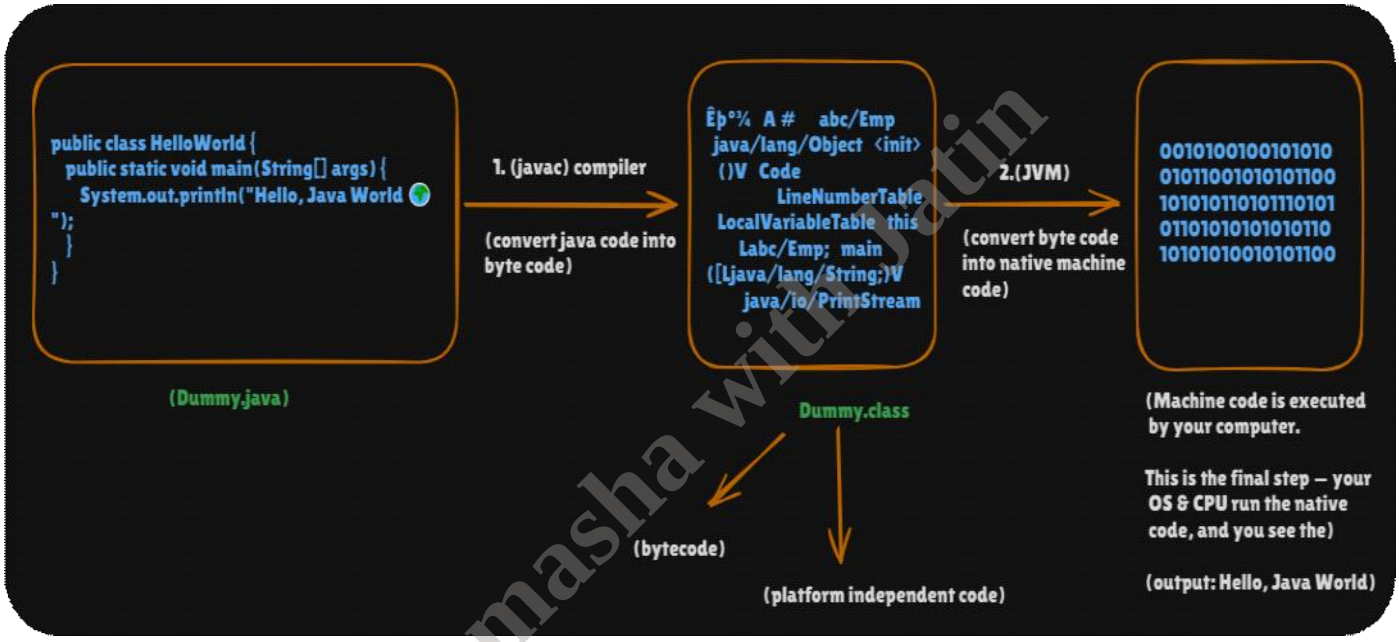
6	}	End of main() method.
7	}	End of class block.

- **Real-Life Analogy**

Think of this program like:

- **Class = TV Show**
- **main() = Entry point where the camera starts rolling**
- **System.out.println = The host speaking to the audience**

⇒ Module 1.6 – Understanding Compilation and Execution in Java (javac + java)



Question:
If ultimately the CPU or OS runs the native machine code...
then why do we say "JVM runs the Java program"?

Short Answer:
Because the JVM is responsible for managing the entire lifecycle of a Java program – from loading bytecode to converting it to machine code and finally coordinating its execution.

👉 **So yes, the CPU physically runs the program,but only because the JVM made it possible.**

Real-Life Analogy: Movie Subtitle Translator
Imagine you're watching a Korean movie, but you only understand English.

The movie = Bytecode

You = CPU

Translator (real-time) = JVM

Subtitle generator = JIT compiler

Even though you (CPU) are watching, you only understand it because the translator (JVM) translates it in real time.

So you say:
"I watched the movie because the translator helped me."

Similarly:
"I ran the Java program because the JVM handled everything."

	Term	Meaning
1	**Machine Code**	Any **binary (0s and 1s)** code that a CPU can execute
2	**Native Machine Code**	Machine code that is **specifically compiled for your system's CPU and OS**

✧ Beginner-Level Questions & Answers

No.	Question	Answer
1	What is Java?	Java is a high-level, object-oriented programming language developed by Sun Microsystems (now Oracle) that allows developers to write code once and run it anywhere.
2	Why is Java called platform-independent?	Java code is compiled into bytecode, which can run on any platform using the JVM — regardless of operating system or hardware.
3	Difference between JDK, JRE, and JVM?	**JDK** = Java Development Kit (for developers), **JRE** = Java Runtime Environment (for users), **JVM** = Java Virtual Machine (runs bytecode).
4	What is bytecode?	Bytecode is an intermediate, platform-independent code generated by the Java compiler and executed by the JVM.
5	What is the role of the JVM?	The JVM loads bytecode, verifies it, converts it to machine code (via interpreter or JIT), and executes it.
6	Compiler vs Interpreter?	Compiler translates code to bytecode at once; interpreter executes line-by-line. JVM uses both (JIT + Interpreter).
7	File extension of compiled Java code?	.class
8	Why must class name match file name?	Because the JVM looks for the public class with the same name as the filename.
9	What is System.out.println()?	A method used to print output to the console. System is a class, out is a static PrintStream object, println() is a method.
10	Purpose of main() method?	It's the entry point of any Java application; the JVM starts execution from here.
11	What happens if we delete the main method?	The JVM will throw a runtime error: Main method not found.
12	Can Java run without a class?	No. Java is class-based. All code must be written inside a class.
13	Meaning of public static void main(String[] args)?	public = accessible by JVM, static = no object needed, void = returns nothing, main = entry point, String[] args = command-line args.
14	How to run Java manually?	Write code → Compile with javac → Run with java
15	Difference between running Java from terminal vs IDE?	Terminal requires manual compilation and running; IDE automates it in the background.

✧ Intermediate-Level Questions & Answers

No.	Question	Answer
1	Explain JVM architecture.	JVM consists of: Class Loader, Bytecode Verifier, Runtime Memory Areas (Heap, Stack, Method Area), Execution Engine (Interpreter + JIT), and Native Interface.
2	JVM memory areas?	1. Method Area (class data), 2. Heap (objects), 3. Stack (method calls), 4. PC Register (current instruction), 5. Native Method Stack.
3	Class loader role?	Loads .class files into JVM memory: Bootstrap, Extension, and Application class loaders.
4	Bytecode verifier?	Checks for illegal code (e.g., stack overflows, access violations) before execution.
5	What is JIT compiler?	JIT = Just-In-Time compiler. It compiles frequently used bytecode into native code at runtime for better performance.
6	How does JVM provide security?	Through bytecode verification, memory management, sandboxing, and restricted class loading.
7	Is Java interpreted or compiled?	Both. Java is compiled to bytecode, then interpreted or JIT-compiled at runtime.
8	What is Garbage Collection?	Automatic removal of unused objects from memory to prevent memory leaks.
9	Stack vs Heap?	Stack = method calls & local variables (LIFO); Heap = objects & global data (shared).
10	Multiple classes in one .java file?	Yes, but only one can be public and must match the file name.
11	Two classes with main() in one	Both can have main(), but only one will run depending on the class you execute.

	file?	
12	◆ Java vs C++/Python in compilation?	◆ Java compiles to bytecode and runs on JVM; C++ compiles to native code; Python is interpreted.
13	◆ JVM steps when running a Java program?	◆ Load class → Verify bytecode → Allocate memory → Execute code (Interpreter/JIT)
14	◆ Can Java run without JDK?	◆ Yes, but only if compiled. You need **JRE + JVM** to run; JDK is for development.
15	◆ Tools like javac, javap?	◆ javac compiles Java code. javap disassembles bytecode for inspection.

✧ Expert-Level Questions & Answers

No.	◆ Question	◆ Answer
1	◆ Full lifecycle of a Java program?	◆ .java → javac → .class → JVM verifies → JIT compiles → CPU executes native code.
2	◆ How does JVM manage memory?	◆ Divides it into areas (heap, stack, etc.), allocates memory dynamically, and collects garbage.
3	◆ What happens during java HelloWorld?	◆ JVM loads HelloWorld.class, verifies it, sets up memory, uses interpreter/JIT, and runs code.
4	◆ How does JIT improve speed?	◆ By compiling hot code paths into native code, skipping interpretation repeatedly.
5	◆ Interpreter vs JIT?	◆ Interpreter = slow but simple (line-by-line), JIT = faster (native compilation at runtime).
6	◆ JIT vs AOT?	◆ AOT (Ahead-of-Time) compiles bytecode to native before runtime; JIT does it during runtime.
7	◆ Classloader delegation model?	◆ Class loaders delegate class loading upward before loading it themselves (parent-first model).
8	◆ Types of Classloaders?	◆ Bootstrap (loads core Java), Extension (optional libs), System (user-defined classes).
9	◆ Bytecode security?	◆ Bytecode verifier, sandboxing, strict access rules prevent harmful operations.
10	◆ Why is Java secure?	◆ Due to no direct memory access, bytecode verification, and controlled class loading.
11	◆ Native code & who generates it?	◆ Native machine code is generated by JIT compiler at runtime.
12	◆ Can Java skip bytecode?	◆ Yes, using tools like GraalVM (AOT compilation), but standard Java uses bytecode.
13	◆ Tools for inspecting bytecode?	◆ javap, JConsole, VisualVM, JProfiler
14	◆ What is a Java Agent?	◆ A tool to modify bytecode at runtime for logging, monitoring, or instrumentation.
15	◆ JVM & multithreading?	◆ JVM uses thread stacks, monitors, and native threads to manage concurrency safely.
16	◆ What is a memory leak in Java and how do you handle it?	◆ A memory leak occurs when unused objects remain referenced and can't be garbage collected — 1.handle it by removing unused references, 2. closing resources, 3.avoiding static-heavy objects, 4.using weak references, 5.detecting leaks with tools like VisualVM or JProfiler.